— Traveling Salesman

k-Coloring: color vertices with k colors so that no two adjacent vertices are the same color.

"بله"

THE

ALGORITHM DESIGN MANUAL

2nd

Steven S. Skiena

ch1 - ch18

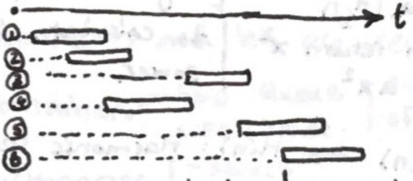
17 pages

2016,11,10 - 2016,11,13

algorithm: a procedure that takes an arbitrary instance of the problem and returns an instance of desired answers.

▶ In the industry, usually a good algorithm is what gets the job done.

* traveling salesman (robot arm) is difficult.



▶ sort by completion date, select by earliest and remove overlaps, until nothing else remains.

The problem with "War & Peace" is that it is too long...

■ = QED: quod erat demonstrandum.

expressing algorithms
- Human language (e.g. English)
- Programming language (e.g. C/C++/Java)
- Pseudo code

Proving Incorrectness via Counter Examples
- think small: target specific shortcomings
- think exhaustively
- hunt for the weakness
- go for a tie: all inputs are the same
- seek extremes

RAM MODEL (Random Access Machine)
- simple operations are constant time
- memory access is constant time
- loops et al are not simple.

▶ also talks about Big-O: see $\boxed{\text{CLRS (4)}}$

$n! \gg 2^n \gg n^3 \gg n^2 \gg n.\log n \gg n \gg \sqrt{n} \gg \log n \gg 1$

```
Power (a, n):
  if (n=0) return 1
  x = power(a, ⌊n/2⌋)
  if n is even return x^2
  else return a x^2
```

uses only $\log(n)$ multiplications to calculate the power.

$\sum_{i=1}^{n} 1/i \approx \ln(n) \longrightarrow H(n)$: Harmonic series

esoteric functions
- $\alpha(n)$: inverse Ackerman's: less than 5 for largest writable integer
- $2^c n^{1+1/c} \leadsto O(n^{1+\epsilon})$

$\longrightarrow f(n)$ dominates $g(n)$ iff: $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

- abstract data types define the contract
  ↳ replacing implementations does not affect the correctness of the program.

Data Structures
1. contiguous: single slab of memory.
2. linked: need to use pointers.

日本 houses in Japan are numbered in the order in which they were built!

o arrays: - constant access time
- space efficiency
- locality
- fixed size → dynamic array 2n effort

→ cell phones are pointers to their owners as they move about

o linked lists → don't overflow unless memory is full.
↳ simpler insertion & deletion.
↳ easier to reallocate items.

▸ Both arrays and lists are recursive data types

o containers → FIFO : Queue / → LIFO : Stack } access irrelevant of content

o dictionaries { - search - insert - delete } operations require a specific key

↳ optionally { - max/min - predecessor/successor

▸ talks about binary search trees: see $\boxed{\text{CLRS(18)}}$

• greedy heuristics tend to grab the best possible thing first.

▸ talks about hashing: see $\boxed{\text{CLRS(14)}}$

▸ talks about Rabin-Karp: see $\boxed{\text{CLRS(49)}}$

uses of hashing {
- duplicate detection
- anti-plagiarism: hash sentences
- pre-transmitting the hash (checksum)

- A quarter of all mainframe cycles were spent sorting ← Knuth

! sorting the data is one of the first things any algorithm designer should think about.

- we should abstract comparison into a function
- SELECTIONSORT: see CLRS(4)
- INSERTIONSORT: see CLRS(3)
- HEAPSORT: see CLRS(9)
- MERGESORT: see CLRS(3) → doesn't need random access → can use linked lists
  - ↳ If we use linked-lists, we can merge without a buffer by rearranging pointers
- QUICKSORT: see CLRS(10)
- BUCKET SORT: see CLRS(12)
  - ↳ great when distribution of data is uniform across some feature of data

- k-way merge sort with a min-heap (PQ) for managing the k-way merge is great for sorting large files (PQ sits in memory while sorted blocks go to disk).

- binary search and problem space halving techniques lie at the heart of divide & conquer solutions.

Graphs

- directed / undirected
- weighted / unweighted
- simple / nonsimple (multi-edge; self loop)
- sparse / dense
- cyclic / acyclic
- embedded / topological
- implicit / explicit → construct on-the-fly
- labeled / unlabeled

representation
- adjacency matrix
- adjacency list
- list of edges

most of the time this is the right answer ⇒ initialization of the data structure can become a bottleneck!

▸ talks about BFS: see CLRS(32)

↳ we can use BFS to discover a two-coloring of graph G ⇒ if exists, G is bipartite

▸ talks about DFS: see CLRS(32) → if we store exploration agenda in a stack instead of a queue we have a DFS.

→ we mark the start & finish "time" of exploration for a node, which sandwiches the exploration of its descendants.

$\frac{\Delta t}{2}$ = # of descendant nodes. edges / tree, back

always point to an anscestor

DFS tree of G

root cut node: $\delta(root) > 1$

bridge cut node: earliest ancestor of $v = v$, $\delta(v) > 0$

parent cut node: earliest ancestor of v is parent of v

articulation points: see CLRS(33)

▶ talks about topological sorting: see CLRS(32)

(ch6)

▶ Talks about MINIMUM SPANNING TREE: see CLRS(34)

Prim: $O(\overset{|E|}{m} + \underset{|V|}{n} \lg n)$ using PQ

Kruskal: $O(m, n)$: $O(m \log m)$ using UNION FIND

Another name for DISJOINT SETS: see CLRS(31)

▶ Talks about Dijkstra: see CLRS(35)

▪ Dijkstra for vertex edges: set weigh (i,j) to $w(j)$

▶ Talks about FLOYD-WARSHALL: see CLRS(36)

▶ Talks about NETWORK FLOW & BIPARTITE MATCHING see CLRS(37-38)

● Try to model problems as graphs & apply known algorithms.

(ch7)

1 million → permutations of 10-11 objects
→ subsets of combinations of 20 items

▷ employing BACKTRACKING is the same as doing a DFS of problem space, with vertices being states, and edges being transitions.

- we can construct all subsets of set of size $|k|$ by promoting the states of presence & absence of item "i" to candidacy. We are done when we have a vector of size $|k|$.
- PRUNING: stopping the search down a path as soon as we have realized this path will not yield an answer.
  ↳ proper pruning can speed up algorithms more significantly than any other factor.

Heuristic Approaches

⇒ only consider parts of the solution space

- All heuristic approaches need to evaluate the "cost/score" of a solution to judge which one is better.

1. RANDOM SAMPLING: generate possible solutions at random, and return the one that looks the best.
   ↳ ⚠ requires the ability to generate random numbers from a unified distribution.

↳ Monte Carlo method

→ Hill Climbing

2 LOCAL SEARCH is the process of picking one solution, then slightly tweaking it until no improvements are possible → great at finding the local optima

Not so great in a landscape full of hills!

3 SIMULATED ANNEALING start at a given temperature. At each step consider "i" neighboring states. Transition if the state is better or a random function tells us to do so. Continue until no further change is found and lower the temperature before the next set of iterations.

↳ We want to avoid getting stuck in local optima by allowing jumps to states that don't improve the solutions. As the temperature decreases, we are less likely to take such risks and will eventually settle on a local optima that is within a larger radius from the starting point than what local searching would allow.

4 GENETIC ALGORITHMS model transitions as mutations and allow for evolution and natural selection to kill off harmful mutations.

↳ Often very slow convergence and also extremely complicated to implement.

PARALLELIZATION does speed up computation but adds complexity to code, is hard to debug, and usually can be beaten by some improvents to the sequential solution.

→ Proper load balancing goes a long way in improving parallelization performance

---

ch8

1 DYNAMIC PROGRAMMING is essentially a trade-off of space for time.

⟹ naturally left-to-right problem spaces
- character strings
- rooted trees
- polygons
- integer sequences

calculating FIBONACCI series

DP

① Come up with the optimizing recursion

② construct bottom-up solution in the right order

Sample Problems

- edit distance
- longest increasing sequence
- morphing images by choosing the best edit for rows of pixels
- partitioning set S into k subsets such that the sum of these S_i are as close to each other as possible.

the cost of each partitioning scheme is the largest sum among its parts. We seek to minimize this.

- parsing context free grammars:

can S_{ij} be parsed as rule $X \to YZ$?

$$M[i,j,X] = \bigvee_{(X \to YZ) \in G} \left(\bigvee_{i=k}^{j} M[i,k,Y] \wedge M[k+1,j,Z] \right)$$

- finding parse errors without breaking the process:

minimum # of parse errors when parsing S_{ij} to $X \to YZ$

$$M'[i,j,X] = \min_{(X \to YZ) \in G} \left(\min_{i=k}^{j} \left(M'[i,k,Y] + M'[k+1,j,Z] \right) \right)$$

- triangulating a polygon so that triangles have minimum total circumference.

When does it work?

→ the recurrence has to be correct

→ the number of remembered partial solutions is small

→ each partial solution is not too difficult to calculate

▶ By showing that we can solve A by converting it to B which runs in $O(f)$ in $O(g)$ conversion time, we reduce A to B.

$\Rightarrow$ ① We have $A \in O(f+g)$
② if $A \in \Omega(h) \Rightarrow B \in \Omega(h-g)$, otherwise the reduction would contradict $A \in \Omega(h)$

→ this is what we use to show hardness.

▶ If we convert an optimization to k to a decision of feasibility of k, we can do binary search to answer the optimization.

$w(u) = \begin{cases} 1 & u \in E \\ 2 & u \notin E \end{cases}$ TSP(G, n)

Hamiltonian cycle $\xrightarrow{\text{reduce}}$ Traveling salesman

$\underbrace{\text{is hard} \quad \text{therefore} \quad \text{is hard}}$

This is the general formulation for NP proofs.

→ We don't even need to know how the target problem works. We want to know if it is difficult or not.

- Cook's theorem: all problems in NP are the same in terms of difficulty.
- NP-hard : not verifiable in polynomial time

1. Do I really understand the problem?
   - → reiterate the problem
   - → articulate all the knowns
2. Can I find a simple heuristic for my problem? → how important is it to solve optimally?

3. Are there special cases of the problem that I know how to solve? → can I generalize my answer?
4. Which algorithmic paradigm is most relevant to my current problem?

The Catalog

Dictionaries

- how many items?
- operation distribution?

→ unsorted linkedlists / arrays
→ sorted linked lists / arrays
→ hash tables
→ BSTs
→ BTrees
→ Skiplists

Priority Queues

great when we know the range of keys

- additional operations?
- is maximum size known?
- can priorities change?

→ sorted array/list
→ binary heap
→ bounded height PQs
→ BSTs
→ Fibonacci heaps

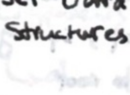
Suffix Trees

→ a graph that represents all the suffixes of a string

Graph Data Structures

- how big? $|V|$
- how dense? $|E|$
- for which algorithms?
- will be modified?

→ matrices
→ adjacency lists
→ edge lists



Set Data Structures

→ bit vectors
→ containers/dictionaries
→ bloom filters: use k hash functions and set bits $H_i(e)$ for inserting e
→ collection of containers
→ generalized bit vector: array where $A[i]$ has the name of subset
→ dictionary with subset attribute
↳ the union-find data structure

Random Number Generation

- Should I have the same random sequence?
- Third-party implementations?
- Implement myself?
- Size of the number?
- Distribution?

Knapsack Problem

→ NP-Complete
→ Budget/Cost
→ Dynamic Programming/Backtracking
⇒ Integer partitioning is a special case of this problem.

SORTING

- range limited? bit vector, counting
- uniformly distributed? buckets.
- sortedness is high?
- external? use in-order of a B-Tree, or use k-way merge sort.

Searching

→ sequential : good up to 20
→ binary : more than 100
- exploit key frequencies : optimal binary search tree using DP by minimizing expected search cost
- self-organizing lists: bump-to-top, splay trees: BST where k gets rotated to root
- one-sided binary search
- external memory? B-Tree, van Emde Boas

Selection

→ naive: $O(n \lg n)$, best: worst-case $O(n)$

→ if we see each element only once, we retain a random sample, & work on that

→ finding the mode is $\Omega(n \lg n)$ ← element uniqueness

Generating Permutations

↳ do we have duplicate items?

→ All: use backtracking

→ Random: use the Knuth method

→ Next: ① Use rank/unrank where $unrank(rank(p), n) \equiv p$

② Use incremental swapping by next()/previous()

Subset generation

→ use grey code

→ use binary counting

Connected Components

↳ also useful for finding weakest point of a graph

→ undirected: there is a path between every two vertices (BFS / DFS)

→ directed ① weak: for (u, v) there is either $(u \rightsquigarrow v)$ or $(v \rightsquigarrow u)$

② strong: there is a path between every pair of edges

→ largest $S \subseteq V$ that forms a complete graph

→ find maximal by sorting according to δ.

Clique → find a dense subgraph instead

→ 2-coloring: discovering bipartite components → DFS

Vertex Coloring → finding chromatic number of graph is NP-complete.

- Vizing's theorem: if Δ max $= k$, we can edge-color with $\Delta+1$.

Edge Coloring ↳ This offers an $O(\Delta|V||E|)$ algorithm for finding $(\Delta+1)$-coloring of the graph

- Can it be lossy?
- Can I simplify? (Burrows-Wheeler)
- Does it have to be real-time?

Text Compression

→ Huffman codes: :(two scans
:(not adaptive

→ Lempel-Ziv (including LZW): build index as we go and reuse frequent blocks

Programming

Pearls 2a

Jon Bentley

col 1 - col 15

8 [illegible]

[illegible] - [illegible]